

AD-A150 556    ROSS: AN OBJECT-ORIENTED LANGUAGE FOR CONSTRUCTING  
SIMULATIONS(U) RAND CORP SANTA MONICA CA  
D MCARTHUR ET AL. DEC 84 RAND/R-3160-RF

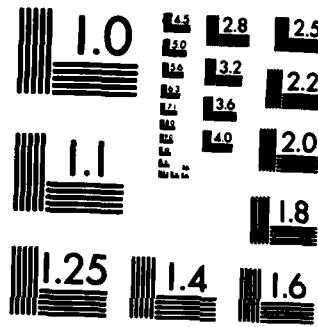
1/1

UNCLASSIFIED F49620-82-C-0018

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS-1963-A

AD-A150 556

85 02 11 104



REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER  R-3160-AF	2. GOVT ACCESSION NO.  AD-A150556	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle)  ROSS: An Object-Oriented Language for Construction Simulations	5. TYPE OF REPORT & PERIOD COVERED  Interim	
7. AUTHOR(s)  D. McArthur, P. Klahr, S. Narain	6. PERFORMING ORG. REPORT NUMBER  F49620-82-C-0018	
9. PERFORMING ORGANIZATION NAME AND ADDRESS  The Rand Corporation 1700 Main Street Santa Monica, CA 90406	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
11. CONTROLLING OFFICE NAME AND ADDRESS  Requirements, Programs and Studies Group (AF/RDQM) Ofc, DSC/R&D And Acquisition Hq USAF, Washington, DC 20330	12. REPORT DATE  December 1984	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	13. NUMBER OF PAGES  19	
	15. SECURITY CLASS. (of this report)  Unclassified	
	16a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report)  Approved for Public Release; Distribution Unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)  No Restrictions		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)  Computerized Simulation Programming Languages Computer Programs Computer Graphics		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number)  see reverse side		

This report provides an overview of ROSS, an object-oriented language currently being developed at Rand. The goal of ROSS is to provide a programming environment in which users can conveniently design, test, and modify large knowledge-based simulations of complex mechanisms. Object-oriented programming languages, and ROSS in particular, enforce a message-passing style of programming in which the system to be modeled is represented as a set of objects and their behaviors (rules for object interaction). This style is especially suited to simulation, since the mechanism or process to be simulated may have a decomposition that maps naturally onto objects, and the real-world interactions between the objects may be easily modeled by object behaviors and object message transmissions. In addition to describing some of the basic ROSS commands and features, the report discusses some software that interfaces directly with ROSS, including a sophisticated screen-oriented editor and a color graphics package. Facilities for browsing among objects and their behaviors are also described, and examples of browsing and editing are presented using SWIRL, a military combat simulation written in ROSS.

2  
R-3160-AF

# **ROSS: An Object-Oriented Language for Constructing Simulations**

David McArthur, Phillip Klahr, Sanjai Narain

December 1984

A Project AIR FORCE report  
prepared for the  
United States Air Force



DTIC  
SELECTED  
FEB 22 1985  
S D  
A

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

## PREFACE

As part of a concept-development effort within the Force Employment Program of Project AIR FORCE, Rand has been investigating techniques to improve computer technology for military combat simulations. This work has focused on developing a research environment that aids users in designing, building, understanding, and modifying battle simulations in order to analyze and evaluate various outcomes. Results of this work include the development of an English-like, object-oriented simulation language called ROSS; the design and implementation of a strategic air penetration simulation built in ROSS, called SWIRL; the design and implementation of a tactical ground-based combat simulation built in ROSS, called TWIRL; and the design of a parallel processing computer architecture for distributing simulations to significantly enhance their performance. These systems are reported in the following Rand publications:

*The ROSS Language Manual*, by D. McArthur and P. Klahr, N-1854-AF, September 1982.

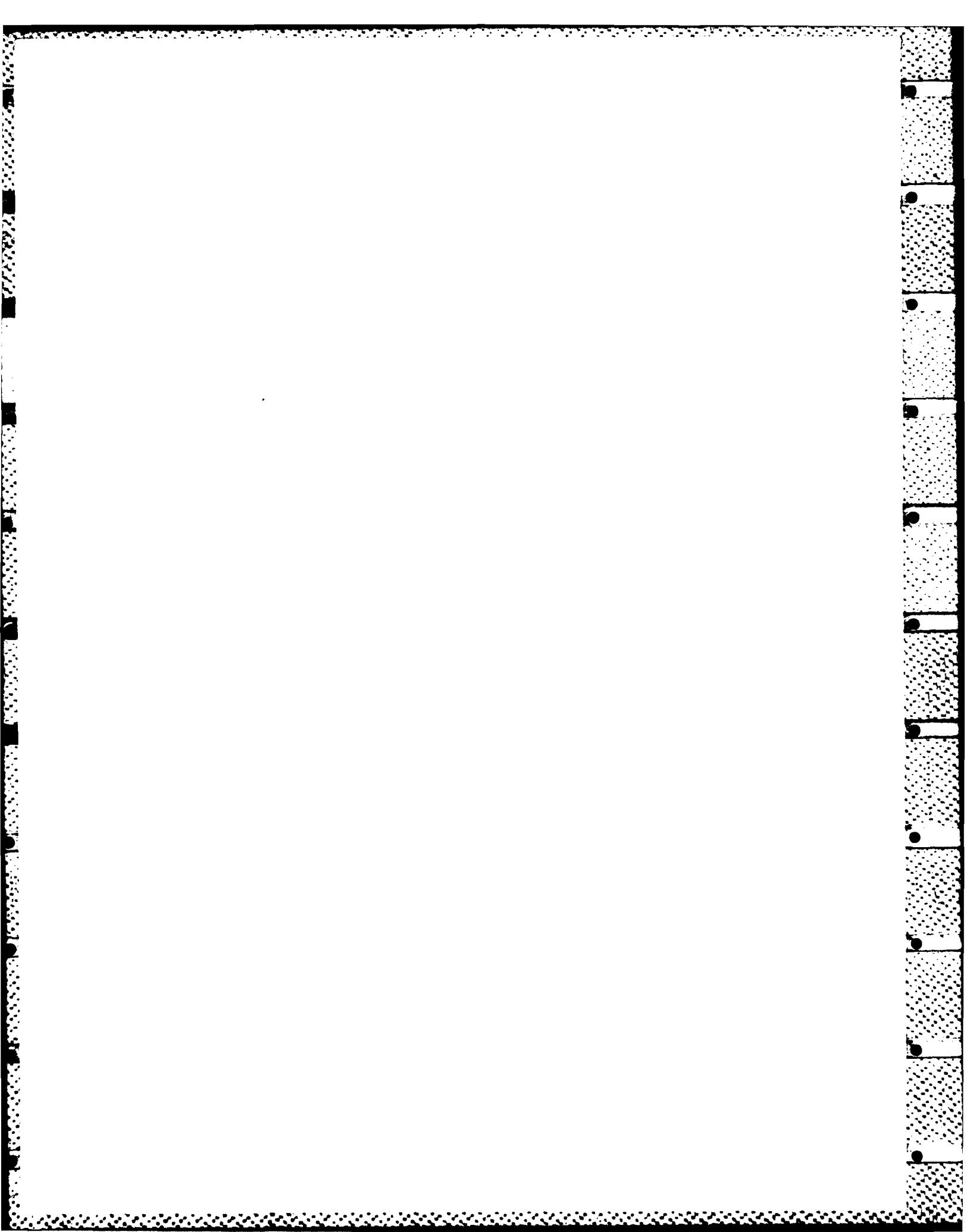
*SWIRL: Simulating Warfare in the ROSS Language*, by P. Klahr, D. McArthur, S. Narain, and E. Best, N-1885-AF, September 1982.

*TWIRL: Tactical Warfare in the ROSS Language*, by P. Klahr, J. Ellis, W. Giarla, S. Narain, E. Cesar, and S. Turner, R-3158-AF, October 1984.

*Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control*, by D. Jefferson and H. Sowizral, N-1906-AF, December 1982.

This report provides an introduction to the ROSS language.

Accession For	
NTIS GRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Announced	<input type="checkbox"/>
Classification	
Comments	
1. DATE RECEIVED	2. DATE INDEXED
3. DATE SERIALIZED	4. DATE FILED
5. FILE NUMBER	6. INDEX NUMBER
7. DATE OF PREP.	8. DATE OF APPROVAL
9. APPROVING OFFICIAL	10. APPROVING OFFICIAL
11. APPROVING OFFICIAL	
12. APPROVING OFFICIAL	
13. APPROVING OFFICIAL	
14. APPROVING OFFICIAL	
15. APPROVING OFFICIAL	
16. APPROVING OFFICIAL	
17. APPROVING OFFICIAL	
18. APPROVING OFFICIAL	
19. APPROVING OFFICIAL	
20. APPROVING OFFICIAL	
21. APPROVING OFFICIAL	
22. APPROVING OFFICIAL	
23. APPROVING OFFICIAL	
24. APPROVING OFFICIAL	
25. APPROVING OFFICIAL	
26. APPROVING OFFICIAL	
27. APPROVING OFFICIAL	
28. APPROVING OFFICIAL	
29. APPROVING OFFICIAL	
30. APPROVING OFFICIAL	
31. APPROVING OFFICIAL	
32. APPROVING OFFICIAL	
33. APPROVING OFFICIAL	
34. APPROVING OFFICIAL	
35. APPROVING OFFICIAL	
36. APPROVING OFFICIAL	
37. APPROVING OFFICIAL	
38. APPROVING OFFICIAL	
39. APPROVING OFFICIAL	
40. APPROVING OFFICIAL	
41. APPROVING OFFICIAL	
42. APPROVING OFFICIAL	
43. APPROVING OFFICIAL	
44. APPROVING OFFICIAL	
45. APPROVING OFFICIAL	
46. APPROVING OFFICIAL	
47. APPROVING OFFICIAL	
48. APPROVING OFFICIAL	
49. APPROVING OFFICIAL	
50. APPROVING OFFICIAL	
51. APPROVING OFFICIAL	
52. APPROVING OFFICIAL	
53. APPROVING OFFICIAL	
54. APPROVING OFFICIAL	
55. APPROVING OFFICIAL	
56. APPROVING OFFICIAL	
57. APPROVING OFFICIAL	
58. APPROVING OFFICIAL	
59. APPROVING OFFICIAL	
60. APPROVING OFFICIAL	
61. APPROVING OFFICIAL	
62. APPROVING OFFICIAL	
63. APPROVING OFFICIAL	
64. APPROVING OFFICIAL	
65. APPROVING OFFICIAL	
66. APPROVING OFFICIAL	
67. APPROVING OFFICIAL	
68. APPROVING OFFICIAL	
69. APPROVING OFFICIAL	
70. APPROVING OFFICIAL	
71. APPROVING OFFICIAL	
72. APPROVING OFFICIAL	
73. APPROVING OFFICIAL	
74. APPROVING OFFICIAL	
75. APPROVING OFFICIAL	
76. APPROVING OFFICIAL	
77. APPROVING OFFICIAL	
78. APPROVING OFFICIAL	
79. APPROVING OFFICIAL	
80. APPROVING OFFICIAL	
81. APPROVING OFFICIAL	
82. APPROVING OFFICIAL	
83. APPROVING OFFICIAL	
84. APPROVING OFFICIAL	
85. APPROVING OFFICIAL	
86. APPROVING OFFICIAL	
87. APPROVING OFFICIAL	
88. APPROVING OFFICIAL	
89. APPROVING OFFICIAL	
90. APPROVING OFFICIAL	
91. APPROVING OFFICIAL	
92. APPROVING OFFICIAL	
93. APPROVING OFFICIAL	
94. APPROVING OFFICIAL	
95. APPROVING OFFICIAL	
96. APPROVING OFFICIAL	
97. APPROVING OFFICIAL	
98. APPROVING OFFICIAL	
99. APPROVING OFFICIAL	
100. APPROVING OFFICIAL	
101. APPROVING OFFICIAL	
102. APPROVING OFFICIAL	
103. APPROVING OFFICIAL	
104. APPROVING OFFICIAL	
105. APPROVING OFFICIAL	
106. APPROVING OFFICIAL	
107. APPROVING OFFICIAL	
108. APPROVING OFFICIAL	
109. APPROVING OFFICIAL	
110. APPROVING OFFICIAL	
111. APPROVING OFFICIAL	
112. APPROVING OFFICIAL	
113. APPROVING OFFICIAL	
114. APPROVING OFFICIAL	
115. APPROVING OFFICIAL	
116. APPROVING OFFICIAL	
117. APPROVING OFFICIAL	
118. APPROVING OFFICIAL	
119. APPROVING OFFICIAL	
120. APPROVING OFFICIAL	
121. APPROVING OFFICIAL	
122. APPROVING OFFICIAL	
123. APPROVING OFFICIAL	
124. APPROVING OFFICIAL	
125. APPROVING OFFICIAL	
126. APPROVING OFFICIAL	
127. APPROVING OFFICIAL	
128. APPROVING OFFICIAL	
129. APPROVING OFFICIAL	
130. APPROVING OFFICIAL	
131. APPROVING OFFICIAL	
132. APPROVING OFFICIAL	
133. APPROVING OFFICIAL	
134. APPROVING OFFICIAL	
135. APPROVING OFFICIAL	
136. APPROVING OFFICIAL	
137. APPROVING OFFICIAL	
138. APPROVING OFFICIAL	
139. APPROVING OFFICIAL	
140. APPROVING OFFICIAL	
141. APPROVING OFFICIAL	
142. APPROVING OFFICIAL	
143. APPROVING OFFICIAL	
144. APPROVING OFFICIAL	
145. APPROVING OFFICIAL	
146. APPROVING OFFICIAL	
147. APPROVING OFFICIAL	
148. APPROVING OFFICIAL	
149. APPROVING OFFICIAL	
150. APPROVING OFFICIAL	
151. APPROVING OFFICIAL	
152. APPROVING OFFICIAL	
153. APPROVING OFFICIAL	
154. APPROVING OFFICIAL	
155. APPROVING OFFICIAL	
156. APPROVING OFFICIAL	
157. APPROVING OFFICIAL	
158. APPROVING OFFICIAL	
159. APPROVING OFFICIAL	
160. APPROVING OFFICIAL	
161. APPROVING OFFICIAL	
162. APPROVING OFFICIAL	
163. APPROVING OFFICIAL	
164. APPROVING OFFICIAL	
165. APPROVING OFFICIAL	
166. APPROVING OFFICIAL	
167. APPROVING OFFICIAL	
168. APPROVING OFFICIAL	
169. APPROVING OFFICIAL	
170. APPROVING OFFICIAL	
171. APPROVING OFFICIAL	
172. APPROVING OFFICIAL	
173. APPROVING OFFICIAL	
174. APPROVING OFFICIAL	
175. APPROVING OFFICIAL	
176. APPROVING OFFICIAL	
177. APPROVING OFFICIAL	
178. APPROVING OFFICIAL	
179. APPROVING OFFICIAL	
180. APPROVING OFFICIAL	
181. APPROVING OFFICIAL	
182. APPROVING OFFICIAL	
183. APPROVING OFFICIAL	
184. APPROVING OFFICIAL	
185. APPROVING OFFICIAL	
186. APPROVING OFFICIAL	
187. APPROVING OFFICIAL	
188. APPROVING OFFICIAL	
189. APPROVING OFFICIAL	
190. APPROVING OFFICIAL	
191. APPROVING OFFICIAL	
192. APPROVING OFFICIAL	
193. APPROVING OFFICIAL	
194. APPROVING OFFICIAL	
195. APPROVING OFFICIAL	
196. APPROVING OFFICIAL	
197. APPROVING OFFICIAL	
198. APPROVING OFFICIAL	
199. APPROVING OFFICIAL	
200. APPROVING OFFICIAL	
201. APPROVING OFFICIAL	
202. APPROVING OFFICIAL	
203. APPROVING OFFICIAL	
204. APPROVING OFFICIAL	
205. APPROVING OFFICIAL	
206. APPROVING OFFICIAL	
207. APPROVING OFFICIAL	
208. APPROVING OFFICIAL	
209. APPROVING OFFICIAL	
210. APPROVING OFFICIAL	
211. APPROVING OFFICIAL	
212. APPROVING OFFICIAL	
213. APPROVING OFFICIAL	
214. APPROVING OFFICIAL	
215. APPROVING OFFICIAL	
216. APPROVING OFFICIAL	
217. APPROVING OFFICIAL	
218. APPROVING OFFICIAL	
219. APPROVING OFFICIAL	
220. APPROVING OFFICIAL	
221. APPROVING OFFICIAL	
222. APPROVING OFFICIAL	
223. APPROVING OFFICIAL	
224. APPROVING OFFICIAL	
225. APPROVING OFFICIAL	
226. APPROVING OFFICIAL	
227. APPROVING OFFICIAL	
228. APPROVING OFFICIAL	
229. APPROVING OFFICIAL	
230. APPROVING OFFICIAL	
231. APPROVING OFFICIAL	
232. APPROVING OFFICIAL	
233. APPROVING OFFICIAL	
234. APPROVING OFFICIAL	
235. APPROVING OFFICIAL	
236. APPROVING OFFICIAL	
237. APPROVING OFFICIAL	
238. APPROVING OFFICIAL	
239. APPROVING OFFICIAL	
240. APPROVING OFFICIAL	
241. APPROVING OFFICIAL	
242. APPROVING OFFICIAL	
243. APPROVING OFFICIAL	
244. APPROVING OFFICIAL	
245. APPROVING OFFICIAL	
246. APPROVING OFFICIAL	
247. APPROVING OFFICIAL	
248. APPROVING OFFICIAL	
249. APPROVING OFFICIAL	
250. APPROVING OFFICIAL	
251. APPROVING OFFICIAL	
252. APPROVING OFFICIAL	
253. APPROVING OFFICIAL	
254. APPROVING OFFICIAL	
255. APPROVING OFFICIAL	
256. APPROVING OFFICIAL	
257. APPROVING OFFICIAL	
258. APPROVING OFFICIAL	
259. APPROVING OFFICIAL	
260. APPROVING OFFICIAL	
261. APPROVING OFFICIAL	
262. APPROVING OFFICIAL	
263. APPROVING OFFICIAL	
264. APPROVING OFFICIAL	
265. APPROVING OFFICIAL	
266. APPROVING OFFICIAL	
267. APPROVING OFFICIAL	
268. APPROVING OFFICIAL	
269. APPROVING OFFICIAL	
270. APPROVING OFFICIAL	
271. APPROVING OFFICIAL	
272. APPROVING OFFICIAL	
273. APPROVING OFFICIAL	
274. APPROVING OFFICIAL	
275. APPROVING OFFICIAL	
276. APPROVING OFFICIAL	
277. APPROVING OFFICIAL	
278. APPROVING OFFICIAL	
279. APPROVING OFFICIAL	
280. APPROVING OFFICIAL	
281. APPROVING OFFICIAL	
282. APPROVING OFFICIAL	
283. APPROVING OFFICIAL	
284. APPROVING OFFICIAL	
285. APPROVING OFFICIAL	
286. APPROVING OFFICIAL	
287. APPROVING OFFICIAL	
288. APPROVING OFFICIAL	
289. APPROVING OFFICIAL	
290. APPROVING OFFICIAL	
291. APPROVING OFFICIAL	
292. APPROVING OFFICIAL	
293. APPROVING OFFICIAL	
294. APPROVING OFFICIAL	
295. APPROVING OFFICIAL	
296. APPROVING OFFICIAL	
297. APPROVING OFFICIAL	
298. APPROVING OFFICIAL	
299. APPROVING OFFICIAL	
300. APPROVING OFFICIAL	
301. APPROVING OFFICIAL	
302. APPROVING OFFICIAL	
303. APPROVING OFFICIAL	
304. APPROVING OFFICIAL	
305. APPROVING OFFICIAL	
306. APPROVING OFFICIAL	
307. APPROVING OFFICIAL	
308. APPROVING OFFICIAL	
309. APPROVING OFFICIAL	
310. APPROVING OFFICIAL	
311. APPROVING OFFICIAL	
312. APPROVING OFFICIAL	
313. APPROVING OFFICIAL	
314. APPROVING OFFICIAL	
315. APPROVING OFFICIAL	
316. APPROVING OFFICIAL	
317. APPROVING OFFICIAL	
318. APPROVING OFFICIAL	
319. APPROVING OFFICIAL	
320. APPROVING OFFICIAL	
321. APPROVING OFFICIAL	
322. APPROVING OFFICIAL	
323. APPROVING OFFICIAL	
324. APPROVING OFFICIAL	
325. APPROVING OFFICIAL	
326. APPROVING OFFICIAL	
327. APPROVING OFFICIAL	
328. APPROVING OFFICIAL	
329. APPROVING OFFICIAL	
330. APPROVING OFFICIAL	
331. APPROVING OFFICIAL	
332. APPROVING OFFICIAL	
333. APPROVING OFFICIAL	
334. APPROVING OFFICIAL	
335. APPROVING OFFICIAL	
336. APPROVING OFFICIAL	
337. APPROVING OFFICIAL	
338. APPROVING OFFICIAL	
339. APPROVING OFFICIAL	
340. APPROVING OFFICIAL	
341. APPROVING OFFICIAL	
342. APPROVING OFFICIAL	
343. APPROVING OFFICIAL	
344. APPROVING OFFICIAL	
345. APPROVING OFFICIAL	
346. APPROVING OFFICIAL	
347. APPROVING OFFICIAL	
348. APPROVING OFFICIAL	
349. APPROVING OFFICIAL	
350. APPROVING OFFICIAL	
351. APPROVING OFFICIAL	
352. APPROVING OFFICIAL	
353. APPROVING OFFICIAL	
354. APPROVING OFFICIAL	
355. APPROVING OFFICIAL	
356. APPROVING OFFICIAL	
357. APPROVING OFFICIAL	
358. APPROVING OFFICIAL	
359. APPROVING OFFICIAL	
360. APPROVING OFFICIAL	
361. APPROVING OFFICIAL	
362. APPROVING OFFICIAL	
363. APPROVING OFFICIAL	
364. APPROVING OFFICIAL	
365. APPROVING OFFICIAL	
366. APPROVING OFFICIAL	
367. APPROVING OFFICIAL	
368. APPROVING OFFICIAL	
369. APPROVING OFFICIAL	
370. APPROVING OFFICIAL	
371. APPROVING OFFICIAL	
372. APPROVING OFFICIAL	
373. APPROVING OFFICIAL	
374. APPROVING OFFICIAL	
375. APPROVING OFFICIAL	
376. APPROVING OFFICIAL	
377. APPROVING OFFICIAL	
378. APPROVING OFFICIAL	
379. APPROVING OFFICIAL	
380. APPROVING OFFICIAL	
381. APPROVING OFFICIAL	
382. APPROVING OFFICIAL	
383. APPROVING OFFICIAL	
384. APPROVING OFFICIAL	
385. APPROVING OFFICIAL	
386. APPROVING OFFICIAL	
387. APPROVING OFFICIAL	
388. APPROVING OFFICIAL	
389. APPROVING OFFICIAL	
390. APPROVING OFFICIAL	
391. APPROVING OFFICIAL	
392. APPROVING OFFICIAL	
393. APPROVING OFFICIAL	
394. APPROVING OFFICIAL	
395. APPROVING OFFICIAL	
396. APPROVING OFFICIAL	
397. APPROVING OFFICIAL	
398. APPROVING OFFICIAL	
399. APPROVING OFFICIAL	
400. APPROVING OFFICIAL	
401. APPROVING OFFICIAL	
402. APPROVING OFFICIAL	
403. APPROVING OFFICIAL	
404. APPROVING OFFICIAL	
405. APPROVING OFFICIAL	
406. APPROVING OFFICIAL	
407. APPROVING OFFICIAL	
408. APPROVING OFFICIAL	
409. APPROVING OFFICIAL	
410. APPROVING OFFICIAL	
411. APPROVING OFFICIAL	
412. APPROVING OFFICIAL	
413. APPROVING OFFICIAL	
414. APPROVING OFFICIAL	
415. APPROVING OFFICIAL	
416. APPROVING OFFICIAL	
417. APPROVING OFFICIAL	
418. APPROVING OFFICIAL	
419. APPROVING OFFICIAL	
420. APPROVING OFFICIAL	
421. APPROVING OFFICIAL	
422. APPROVING OFFICIAL	
423. APPROVING OFFICIAL	
424. APPROVING OFFICIAL	
425. APPROVING OFFICIAL	
426. APPROVING OFFICIAL	
427. APPROVING OFFICIAL	
428. APPROVING OFFICIAL	
429. APPROVING OFFICIAL	
430. APPROVING OFFICIAL	
431. APPROVING OFFICIAL	
432. APPROVING OFFICIAL	
433. APPROVING OFFICIAL	
434. APPROVING OFFICIAL	
435. APPROVING OFFICIAL	
436. APPROVING OFFICIAL	
437. APPROVING OFFICIAL	
438. APPROVING OFFICIAL	
439. APPROVING OFFICIAL	
440. APPROVING OFFICIAL	
441. APPROVING OFFICIAL	
442. APPROVING OFFICIAL	
443. APPROVING OFFICIAL	
444. APPROVING OFFICIAL	
445. APPROVING OFFICIAL	
446. APPROVING OFFICIAL	
447. APPROVING OFFICIAL	
448. APPROVING OFFICIAL	
449. APPROVING OFFICIAL	
450. APPROVING OFFICIAL	
451. APPROVING OFFICIAL	
452. APPROVING OFFICIAL	
453. APPROVING OFFICIAL	
454. APPROVING OFFICIAL	
455. APPROVING OFFICIAL	
456. APPROVING OFFICIAL	
457. APPROVING OFFICIAL	
458. APPROVING OFFICIAL	
459. APPROVING OFFICIAL	
460. APPROVING OFFICIAL	
461. APPROVING OFFICIAL	
462. APPROVING OFFICIAL	
463. APPROVING OFFICIAL	
464. APPROVING OFFICIAL	
465. APPROVING OFFICIAL	
466. APPROVING OFFICIAL	
467. APPROVING OFFICIAL	
468. APPROVING OFFICIAL	
469. APPROVING OFFICIAL	
470. APPROVING OFFICIAL	
471. APPROVING OFFICIAL	
472. APPROVING OFFICIAL	
473. APPROVING OFFICIAL	
474. APPROVING OFFICIAL	
475. APPROVING OFFICIAL	
476. APPROVING OFFICIAL	
477. APPROVING OFFICIAL	
478. APPROVING OFFICIAL	
479. APPROVING OFFICIAL	
480. APPROVING OFFICIAL	
481. APPROVING OFFICIAL	
482. APPROVING OFFICIAL	
483. APPROVING OFFICIAL	
484. APPROVING OFFICIAL	
485. APPROVING OFFICIAL	
486. APPROVING OFFICIAL	
487. APPROVING OFFICIAL	
488. APPROVING OFFICIAL	
489. APPROVING OFFICIAL	
490. APPROVING OFFICIAL	
491. APPROVING OFFICIAL	
492. APPROVING OFFICIAL	
493. APPROVING OFFICIAL	
494. APPROVING OFFICIAL	
495. APPROVING OFFICIAL	
496. APPROVING OFFICIAL	
497. APPROVING OFFICIAL	
498. APPROVING OFFICIAL	
499. APPROVING OFFICIAL	
500. APPROVING OFFICIAL	
501. APPROVING OFFICIAL	
502. APPROVING OFFICIAL	
503. APPROVING OFFICIAL	
504. APPROVING OFFICIAL	
505. APPROVING OFFICIAL	
506. APPROVING OFFICIAL	
507. APPROVING OFFICIAL	
508. APPROVING OFFICIAL	
509. APPROVING OFFICIAL	
510. APPROVING OFFICIAL	
511. APPROVING OFFICIAL	
512. APPROVING OFFICIAL	



## SUMMARY

This report provides an overview of ROSS, an object-oriented language currently being developed at Rand. The goal of ROSS is to provide a programming environment in which users can conveniently design, test, and modify large knowledge-based simulations of complex mechanisms.

Object-oriented programming languages, and ROSS in particular, enforce a *message-passing* style of programming in which the system to be modeled is represented as a set of objects and their *behaviors* (rules for object interaction). This style is especially suited to simulation, since the mechanism or process to be simulated may have a decomposition that maps naturally onto objects, and the real-world interactions between the objects may be easily modeled by object behaviors and object message transmissions.

In addition to describing some of the basic ROSS commands and features, the report discusses some software that interfaces directly with ROSS, including a sophisticated screen-oriented editor and a color graphics package. Facilities for browsing among objects and their behaviors are also described, and examples of browsing and editing are presented using SWIRL, a military combat simulation written in ROSS.

## CONTENTS

<b>PREFACE</b> .....	iii
<b>SUMMARY</b> .....	v
<b>Section</b>	
<b>I. INTRODUCTION</b> .....	1
Simulation as a Reasoning and Design Tool .....	1
Shortcomings of Existing Simulation Techniques .....	2
<b>II. OVERVIEW OF THE ROSS LANGUAGE</b> .....	4
The Structure of Object-Oriented Simulation Models:	
Objects, Messages, and Behaviors .....	5
Object Hierarchies and Inheritance .....	6
Getting Simulations Started: Planning and the Clock .....	9
Advantages and Disadvantages of Object-Oriented Languages for Simulation .....	10
English-like Code in ROSS .....	10
Interactive Browsing and Editing Facilities .....	11
Using ROSS in the Emacs Editor .....	13
Textual Output Facilities .....	14
Graphical Output Facilities .....	17
<b>III. CONCLUSIONS</b> .....	18
<b>BIBLIOGRAPHY</b> .....	19

## I. INTRODUCTION

This report presents an overview of ROSS, an object-oriented simulation language. ROSS is one of the first languages that attempts to marry artificial intelligence (AI) methods with simulation technology. We have found that the marriage benefits both parties. Simulation is a powerful inference and design tool that has been largely overlooked by AI researchers; however, many human experts rely heavily on qualitative simulation to understand and fix complex systems. We believe that incorporating simulations as components would increase the power of many expert systems. But while simulation is potentially a powerful tool for modeling, understanding, and designing complex systems, present-day simulators do not have the necessary features to allow simulation to fulfill its potential. This report discusses these shortcomings and shows how ROSS uses AI techniques to create a simulation environment that overcomes many of the limitations. The report is not a manual providing detailed discussions of the commands or rationale of the ROSS language. Those discussions are provided in McArthur and Klahr (1982), and equally detailed discussions of specific simulations written in ROSS can be found in Klahr et al. (1982a, 1982b) and Klahr et al. (1984). The language and its applications are described here only to the degree necessary to provide a general overview.

### SIMULATION AS A REASONING AND DESIGN TOOL

It is often useful to be able to understand a dynamic system without manipulating it in the real world. Some real-world systems perform too slowly (e.g., economic systems), some have dangerous consequences (e.g., nuclear reactors), and some are awkward or impossible to manipulate directly (e.g., the solar system). In such cases, it is desirable to be able to draw conclusions about the behavior of the system by reasoning with a model of the system, rather than by observing the system itself.

Model-based reasoning has many uses. Most basically, it can be used to understand and predict the performance of the system. When the system is one that can be modified by the user, model-based reasoning can also be used for intervention and design. In some systems, only limited changes are possible (e.g., a medical treatment modifies bodily processes in a constrained way), but in others, the user has full control over structural properties (e.g., the design of a computer). For systems that can be modified, conclusions drawn by reasoning about system behavior can be used as a basis for making coherent structural changes. These changes then lead to an examination of the behavior of the new system, and more rounds of changes. A successful *design cycle* ideally converges on a system that exemplifies some predetermined behavioral properties.

Unfortunately, the intuitive methods currently used to model systems are inaccurate and incomplete, while mathematical (analytic) methods, though accurate, may be of limited value because of the difficulty of formally modeling all the complexities of many dynamic systems. Simulation represents a useful tool for reasoning about the possible behaviors of dynamic

systems which potentially avoids the imprecision of intuition, while providing greater expressive power than mathematics.<sup>1</sup>

## SHORTCOMINGS OF EXISTING SIMULATION TECHNIQUES

While simulation technology is potentially a powerful tool for modeling, understanding, and designing systems, large-scale simulators do not provide the features necessary to allow simulation to fulfill its potential. Four major shortcomings of present simulations and simulators are described below.

### Inability to Verify the Completeness and Accuracy of Models

A large-scale simulator typically contains many types of knowledge that collectively comprise a model of a real-world system. Some types are explicitly represented and understandable, e.g., an object's properties, such as location, velocity, or altitude. Other types of knowledge, such as how objects behave, how they interact with other objects, and how they make decisions, may be impossible to understand. Users may have to search through volumes of manuals, looking for behavioral descriptions that are buried in incomprehensible code. Even if the information can be located and understood, the user cannot be sure that the system actually performs as specified. Much can get lost in the translation to computer code. Further, knowledge is typically not well structured. Embedded assumptions are hidden, scattered, and fragmented throughout the program. The initial structure of the first version of a simulation is often lost as more complexity is added or modifications are made. As a result, there is no assurance that the simulation embeds an accurate or complete model of the dynamic system. The user cannot have confidence in either the predictions or the design advice the simulation might suggest.

### Inability to Modify Models and Construct Alternative Models

Models embedded in simulations cannot be easily modified, for many of the same reasons they cannot be verified for accuracy. If the key behaviors of simulation objects are hidden in masses of code—or worse, distributed across the code—users will not be able to alter them in a coherent fashion. The simulation may thus inhibit rather than promote the easy changes required to provide a good environment in which to investigate alternatives.

### Incomprehensibility of Results

A simulation that is intelligible, and thus can be verified to represent an accurate model, can be confidently expected to produce data that describe the behavior of the modeled dynamic system. However, if the data are not presented effectively, it can be almost impossible to see the most important behavioral properties of the system. For example, large military simulations can generate hundreds of pages of numeric output for each simulation run. Determining the main global features of the simulation's performance from such output is a slow process at best; at worst, important trends may be overlooked entirely.

<sup>1</sup>We do not claim that simulation is the *only* means of modeling a system and drawing useful conclusions about its behavior, or even that it is necessarily the best means. Simulation is just one of many modeling tools; moreover, simulations do not enable users to answer several important kinds of questions about the system being modeled. For a more detailed discussion of the inherent limitations of state-transition simulations, see Davis et al. (1982).

**Long Run Times**

Even simulations that are intelligible and supply the support required to try out a wide range of alternatives are not useful if they are too slow. Certainly, simulations that have reached a final "production" state must execute rapidly. The analyst expects to simulate complex processes in minutes, not hours. On a more subtle level, it is frequently necessary to try out many parameter settings or investigate many alternative models to draw any reliable inferences from a simulation. Such multiple experiments are prohibitive if each run takes more than a few minutes. Unfortunately, many simulations run even slower than real time. (Our approach to speed is discussed in Jefferson and Sowizral (1982), and the speeds of several different implementations of a ROSS simulation called SWIRL are discussed in Narain et al. (1983).)

## II. OVERVIEW OF THE ROSS LANGUAGE

The ROSS language (McArthur and Klahr, 1982) was designed at Rand to overcome some of the shortcomings of existing simulations and to provide a superior environment in which to model, understand, and design dynamic systems. ROSS relies heavily on recently developed AI techniques and expert-systems technology (Hayes-Roth et al., 1983).

ROSS is an *English-like, object-oriented language*. Several examples of its English-like flavor are given later in this report. More detailed examples are presented in Klahr et al. (1984) and especially in Klahr et al. (1982b), which discusses virtually all the code of the SWIRL simulation. The English-like nature of the code makes it readable, and makes the models embedded in it intelligible to users who may not be programming experts. ROSS's object-oriented nature, which is shared by the SMALLTALK (Goldberg and Robson, 1983), PLASMA (Hewitt, 1977), FLAVORS (Weinreb and Moon, 1981), and DIRECTOR (Kahn, 1979) languages, imposes a style of programming that is highly suited to simulation. Because ROSS is interactive (it is implemented in Lisp), a ROSS simulation can be interrupted while it is running, the state can be queried or the code modified, and the simulation can then be resumed. With compiled simulation languages, such as SIMSCRIPT (Kiviat et al., 1968), the user must specify a simulation and let it run to completion before making any modifications. Thus, alternate designs can be explored in ROSS substantially more quickly and easily. ROSS's interactive nature also makes the debugging of simulations much simpler and more rapid. Over the course of constructing large simulations, this can mean substantial savings in development costs.

To make simulation results more comprehensible, ROSS provides a tracing facility that produces textual simulation output. In addition, ROSS is directly linked to a "movie generator" and graphics facility, so that visual representations can be generated as the simulation is running. This visual presentation is invaluable in discerning global trends in simulation performance.

ROSS has been operational for several years and has been implemented in a wide variety of Lisps (Narain et al., 1983), including Maclisp, Interlisp-20, Vax-Interlisp, Interlisp-D, Franz-lisp, and Zetalisp. An earlier ROSS simulation, called SWIRL (Strategic Warfare in the ROSS Language), for example, provides a prototype of a design tool for military strategists in the domain of air battles (Klahr et al., 1982a, 1982b). SWIRL embeds knowledge about offensive and defensive battle strategies and tactics. It accepts from the user a simulation environment representing offensive and defensive forces, and it uses the specifications in its knowledge base to produce a simulation of an air battle. SWIRL also enables the user to observe the progress of the air battle in time, by means of a graphical interface. Exploiting ROSS's ability to easily modify simulation objects and their behaviors, SWIRL encourages the user to explore a wide variety of alternatives and to discover increasingly effective options in offensive and defensive strategies. Below, we elaborate on the main features of ROSS that are used in SWIRL.

## THE STRUCTURE OF OBJECT-ORIENTED SIMULATION MODELS: OBJECTS, MESSAGES, AND BEHAVIORS

Object-oriented simulation languages enforce a style of programming that parallels the way we intuitively think of the processes in the dynamic system we are modeling. Many systems are naturally described in terms of separate components. For example, a car engine includes a carburetor, a transmission, etc. Further, the behavior of the system typically arises as objects interact by transmitting forces or information to one another, or by coming into physical proximity. In ROSS, such systems are modeled in a very natural way by creating objects to model each of the components of the dynamic system, and by modeling the interactions between them as *message passing*.

In ROSS, messages are sent from one object to another, using the following kind of form:<sup>1</sup>

(1) (ask *fighter-base1* send *fighter2* guided by *gci3* to *penetrator4*).

The general syntax for message transmissions is:

(2) (ask *<object>* *<message>*)

where *<object>* is the name of any ROSS object or actor, and *<message>* is any sequence of Lisp atoms defining a legal ROSS message. In (1) above, the object named *fighter-base1* receives the message *send fighter2 guided by gci3 to penetrator4*.

When an object receives a message, it must have a way of responding. The user therefore must define a *behavior* whose *pattern* matches the message and whose *actions* represent the appropriate response. In this case, the behavior might be:

(3) (ask *fighter-base* when receiving  
 (send >*fighter* guided by >*gci* to >*penetrator*)  
 (ask !*myself* schedule after  
 !(ask *myself* recall your scramble-delay) seconds  
 tell !*fighter* chase !*penetrator* guided by !*gci*)  
 (ask !*myself* add !*fighter* to your list of fighters-scrambled)  
 (ask !*myself* remove !*fighter* from your list of  
 fighters-available)).

More generally, behaviors, like all computation in an object-oriented language, are defined by message transmissions of the form:

(4) (ask *<object>* when receiving *<message-template>* *<body>*)

where *<object>* is the name of any ROSS object or actor, *<message-template>* is any sequence of Lisp atoms defining a legal ROSS message, possibly including variables, and *<body>* is any arbitrary ROSS or Lisp code.

Just as ROSS objects are meant to model components of a real-world system, behaviors model the repertoire of ways that particular kinds of objects can respond to different inputs,

<sup>1</sup>Reserved ROSS keywords are shown in boldface.

information, or forces in the real world. Note that behaviors are attached to specific objects; they are not global functions. This captures the notion that different kinds of real-world entities have different behavioral capabilities.

When the message in (1) is sent, *fighter-base1* will *pattern-match* that message (i.e., *send fighter2 guided by gci3 to penetrator4*) against the message templates of all the behaviors it knows, until it finds the appropriate one, in this case, *send >fighter guided by >gci to >penetrator*. This template matches the message because all the words in it are either identical to analogous items in the message or are *variables*. Variables are indicated by the prefixes *>* or *+*, and they match any word in the incoming message. Thus, *>fighter* matches *fighter2*, and the variable *fighter* will be bound to *fighter2* during the execution of the behavior associated with this template. (Similarly, *gci* will be bound to *gci3*, and *penetrator* will be bound to *penetrator4*.) Pattern variables such as *fighter* are prefixed by *!* in the behavior body. This prefix forces the variable to be evaluated, i.e., its value is used, not its name (ROSS is a nonevaluating form of Lisp<sup>2</sup>). The behavior body itself embeds message transmissions, either to the object that received the current message (the object is always the value of the variable *myself*) or to any other object. Thus, a single message transmission can trigger an arbitrarily complex chain of subsequent transmissions.

## OBJECT HIERARCHIES AND INHERITANCE

Another important semantic concept in ROSS is that of object hierarchies and inheritance. In item (1) above, a message was sent to an object called *fighter-base1*, while in (3), the user defined a behavior for an object called *fighter-base*. How can *fighter-base1* use a behavior that was defined for *fighter-base*? The answer is that although *fighter-base1* was not given this behavior directly, it can *inherit* it. To understand how inheritance works in ROSS, we need to understand how objects are created.

Objects can be created by forms such as

(5) (ask *fighter-base* create instance *fighter-base1*).

This message causes *fighter-base1* to be created as an instance of *fighter-base*. Semantically, *fighter-base* should be interpreted as denoting the class of all *fighter-bases*, while *fighter-base1* denotes a particular element of that class. This fundamental distinction between *generic* and *instance* objects is very useful in modeling real-world systems, which have different kinds of components, each instance of which has the same or similar properties. For example, all dogs (at least all healthy, normal dogs) have four legs, and most dogs bark when they see the mailman. A good modeling language should provide a way to make these quantified statements and to infer their truth for any particular dog. By allowing the creation of generic objects, ROSS provides a means of making quantified statements; by allowing inheritance, ROSS enables these statements to be instantiated for individuals.

The procedure of inheritance that models this instantiation is very simple. If an object (e.g., *fighter-base1*) has been created as an instance of a class (e.g., *fighter-base*), when the instance receives a message, it first looks at behaviors that have been explicitly attached to it,

<sup>2</sup>Lisp comes in two basic types: evaluating and nonevaluating. An evaluating (or EVAL) version evaluates arguments to function calls before giving them to the function; a nonevaluating (or EVALQUOTE) version does not. For example, the EVALQUOTE form *(list a b)* and the EVAL form *(list 'a 'b)* both give the same result, namely a list of two elements, *a* and *b*.

to find one that matches the message. If it fails, it will consult the behaviors of the class objects of which it is an instance.

Simple inheritance has been generalized in two ways in ROSS. First, class objects may be subclasses of other objects, just as instances are members of classes. To create a subclass, one might say

(6) (ask fighter-base create generic prop-fighter-base)

(7) (ask fighter-base create generic jet-fighter-base).

This would indicate that there are two different kinds of fighter-bases, those that handle jets and those that do not. The importance of subclassing is that it allows users to express quantified statements of arbitrarily limited scope. There may be some things that all fighter-bases can do, some that only jet bases can do, and some that only prop bases can do.

Subclassing can be complex in ROSS, inducing a class-inclusion object *hierarchy*. An example of the object hierarchy used in SWIRL is shown in Fig. 1. When an instance of a generic object in this hierarchy receives a message, it does an inheritance search up from its location in the leaves of the hierarchy, to find an ancestral object that has a behavior matching the message it has received. Thus when matching an incoming message, *fighter-base1* will search, in order, object behaviors for *fighter-base1*, *fighter-base*, *fixed-object*, and *simulator*.

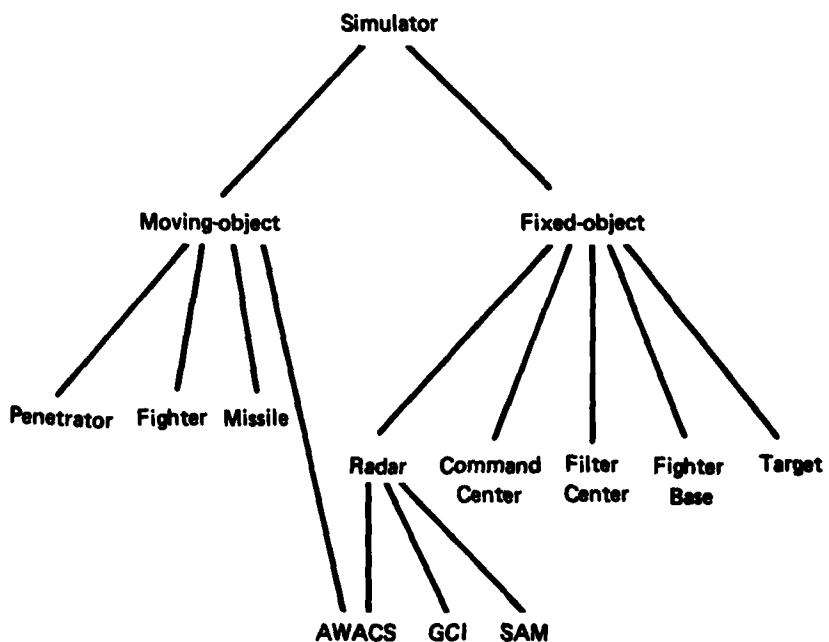


Fig. 1—SWIRL hierarchy of basic objects

The object hierarchy in Fig. 1 is not strictly a tree. AWACS (a type of airborne radar) is a subclass of both *moving-object* and *radar*. This *multiple inheritance* is the second way simple inheritance has been generalized in ROSS. Multiple inheritance was not a feature of earlier object-oriented languages, but it has proven highly useful in our ROSS simulations. Its value derives from the fact that it is often useful to view an object from multiple perspectives, or as being described in several ways, not as a fixed kind of thing. From one perspective, an AWACS can be viewed as having the properties of an aircraft, or a moving object; from another perspective, it can be viewed as a detector of electronic signals, or a radar.

Finally, although we have been talking primarily about the inheritance of behaviors, ROSS objects also possess attributes, and they too can be inherited. The behaviors of ROSS objects model the dynamic actions and reactions of real-world entities, i.e., how they respond to various inputs; the attributes of ROSS objects model the static features or properties of such objects (which may, however, dynamically change during a simulation). For example, we might have created the class object *fighter-base* as follows:

(8) (ask fixed-object create generic fighter-base with

position	(0 0)	;position
status	active	;active or destroyed
filter-center	nil	;each base has one
fighters-available	nil	;list of free fighters
fighters-scrambled	nil	;occupied fighters
fighters-destroyed	nil	;its fighters lost
scramble-delay	10	;time to scramble once told
alert-delay	10	;delay to alert
alert-duration	1800	;how long to remain alert
range	400.0)	;how far its fighters can go

Now, when *fighter-base1* wants to find out its alert-duration, it uses the message transmission

(9) (ask fighter-base1 recall your alert-duration)

to find the value associated with *fighter-base*, effectively making the inference that since all *fighter-bases* have an alert-duration of 1800 seconds, it must have an alert-duration of 1800 seconds. Some of the attributes of *fighter-base* are to be interpreted as defaults, rather than as required values. For example, although all *fighter-bases* have an alert-duration of 1800 seconds, the position of a particular *fighter-base* can only be inferred to be (0 0) if no other information about that base is available, that is, if the base does not have a position explicitly associated with it. Also note that some attributes are known to exist but are not given any default values. These attributes have a *nil* initial value. When *fighter-base* is asked to create instances of itself, values for these attributes must be specified.

ROSS provides commands to create and manipulate the attributes of objects. For example, the message

(10) (ask fighter-base1 set your status to destroyed)

will set the *status* attribute of *fighter-base1* to be *destroyed*. (If the particular attribute does not exist for the object, it is created.) The following message shows an example of modifying an inherited attribute:

(11) (ask *fighter-base1* increment your *alert-duration* by 100).

If *fighter-base1* did not have a value for the attribute *alert-duration*, it would inherit the value of 1800 seconds from the *fighter-base* class to which it belongs, add 100 to it, and incorporate the result, 1900, as the value of its *alert-duration* attribute (without effecting any attributes of *fighter-base*). Attributes can be created and manipulated dynamically within a simulation or by a user at any time.

## GETTING SIMULATIONS STARTED: PLANNING AND THE CLOCK

We have described computation in ROSS, but not simulation. A simulation language must provide a way of modeling events occurring over time. ROSS provides two such facilities. First, ROSS allows any object to plan an event to happen at any future time, using commands of the form

(12) (ask *fighter-base1* plan after 20 seconds  
tell *fighter1* chase *penetrator2* guided by *gci3*).

Once *fighter-base1* has issued this command, ROSS will ensure that the message embedded in the plan is sent to *fighter1*, not immediately, but after 20 seconds. (Of course, "20 seconds" refers to *simulated* real time, not real time or CPU time). Operationally, this means that all messages planned to be sent 19 seconds or less in the future are guaranteed to be sent before this message, and all planned for 21 seconds will be sent after it.

To execute planned events, ROSS provides a clock. The clock is a primitive ROSS object (called *nclock*) that advances simulation time when given commands such as

(13) (ask *nclock* tick).

A "tick" is a variable number of seconds, determined by the *ticksizes* attribute of *nclock*. When the clock ticks, it not only advances simulated real time by a given interval of time, it also sends each of the messages (in time order) that have been planned for the current time interval. For example if (12) then (13) were issued at time 120 with a clock ticksizes of 25 seconds, the simulated real time would first advance to 140 seconds (the time of the next message to be sent in the current time interval); the message embedded in (12), i.e., *tell fighter1 chase penetrator2 guided by gci3*, would be transmitted; and finally, the clock would advance to 145 and end the current tick.

The user can control the size of a tick by giving commands such as

(14) (ask *nclock* set your ticksizes to 20).

Because the user can decide how many ticks to execute at a time and can specify the size of a tick, he has complete control over the "grain" of the simulation. At one extreme, he can let a

simulation run to completion without interruption; at the other, he can stop it after a very short duration, examine the state of any of the simulation objects in detail, and then resume running.

### ADVANTAGES AND DISADVANTAGES OF OBJECT-ORIENTED LANGUAGES FOR SIMULATION

The main advantage of object-oriented languages for simulation is that they immediately suggest a way to view the dynamic system to be modeled. Our experience with SWIRL (Klahr et al., 1982a, 1982b) and TWIRL (Klahr et al., 1984) indicates that an object-oriented style of computation is especially suited to simulation in domains that may be thought of as consisting of autonomous *intentionally* interacting components. In such domains, the programmer can discern a natural mapping of their constituent components onto objects and of their intentional interactions, such as communication, onto message transmissions. Indeed, experts in many domains may find the object-oriented metaphor a natural one around which to organize and express their knowledge (Klahr and Faught, 1980).

However, in developing simulations in ROSS, we have discovered some events, or interactions between real-world objects, that are not as easily modeled as we might like them to be. These events are often side effects of deliberate actions (e.g., a penetrator appearing as a blip on a radar screen is a side effect of the penetrator flying its course and entering a radar range). Such events are important, since they may trigger other actions (e.g., a radar detecting a penetrator and notifying a filter center). However, these *nonintentional* events do not correspond to real-world message transmissions (e.g., a penetrator does not notify a radar that it has entered the radar's range). An important issue in the development of SWIRL has been the problem of capturing these nonintentional events in an object-oriented framework (i.e., via message transmissions). This problem and its various possible solutions are discussed in Klahr et al. (1982b).

### ENGLISH-LIKE CODE IN ROSS

While the style of programming enforced by ROSS's object-oriented nature is the most important factor contributing to the modifiability of ROSS models, the highly readable character of the code is also important. Two features make ROSS code English-like. First, because behavior invocation uses pattern-matching, the user can define arbitrarily verbose behavior templates if he wishes. Thus, in (3) above, the user employed the template

(15) (send >fighter guided by >gci to >penetrator)

rather than the less English-like and less comprehensible

(16) (send >f guided-by >g >p)

even though either is quite acceptable as a ROSS command. If one were to implement behaviors in Lisp as functions, the resulting function calls would about as unintelligible as the code in (16):

(17) (send-guided-by-to fighter gci penetrator).

ROSS allows for even greater readability through its *abbreviations package*. This package was used in the development of SWIRL and TWIRL to generate code that would be comprehensible to strategic and tactical warfare experts who were programming novices. For example, after specifying the following abbreviations:

(18) (abbreviate '(ask !myself) 'you)  
(abbreviate '(ask !myself recall your) 'your)  
(abbreviate '(!) 'the)

we could rewrite (3) as

(19) (ask fighter-base when receiving  
      (send >fighter guided by >gci to >penetrator)  
      ('you schedule after !(your scramble-delay) seconds  
      tell `the fighter chase `the penetrator guided by `the gci)  
      ('you add `the fighter to your list of fighters-scrambled)  
      ('you remove `the fighter from your list of fighters-available)).

The ROSS abbreviations package allows for *personalized English-like programming*, in contrast to languages like ROSIE (Fain et al., 1981), whose English-like syntax is fixed by the language designers. A personalized English syntax enhances the writability of code, whereas languages whose syntax is fixed often prove difficult to use because the user falsely assumes that *his* English syntax is the language's. In ROSS, this problem does not exist, because the user *does* define his own language syntax.

Second, even extensive use of the ROSS abbreviations package carries almost no computational overhead. ROSS expands the abbreviations only once, at load time, somewhat like displacing macros. Other English-like languages may incur high costs in parsing time. In fact, more time can be spent parsing the English-like surface representation into a machine-runnable form than actually executing the runnable code.

## INTERACTIVE BROWSING AND EDITING FACILITIES

ROSS encourages the development of highly structured, interactive *browsing* and *editing* facilities. An interactive browser can be thought of as an interface that is an expert in the structure of a simulation and so can quickly guide even a naive user to any piece of a simulation model—any object or behavior. An interactive editor extends the browser by permitting the user to modify the code, once he has found the part of the simulation model he wishes to change. Collectively, these features facilitate rapid, coherent modification of models by users who are familiar with the *models*, but not necessarily with the specifics of their *implementation*.

Figure 2 shows a user interaction with a simple but effective browser we constructed for the SWIRL simulation. Item (20) is the top-level menu. (Virtually all user interaction with SWIRL involves menu selections.) The user selects option 9, which invokes the browser to guide him through the code. If the user has been interacting directly with ROSS, and not with the SWIRL menu, the browser could have been invoked by "(ask browser help)." The browser is a full-fledged ROSS object. At (21), the browser presents a menu of all SWIRL objects it

## (20) Select option:

- 1—Break into ROSS
- 2—Load compiled SWIRL
- 3—Load interpreted SWIRL
- 4—Recompile interpreted SWIRL files
- 4—Recompile interpreted SWIRL files
- 5—Load a simulation environment
- 6—Run simulation with graphics
- 7—Run simulation without graphics
- 8—Activate historian and reporter
- 9—Browse or edit behaviors
- 10—Exit SWIRL & ROSS 9

## (21) 1.—PENETRATOR

- 2.—FIGHTER
- 3.—GCI
- 4.—SAM
- 5.—AWACS
- 6.—RADAR
- 7.—FILTER-CENTER
- 8.—COMMAND-CENTER
- 9.—FIGHTER-BASE
- 10.—TARGET
- 11.—MISSILE
- 12.—MOVING-OBJECT
- 13.—FIXED-OBJECT
- 14.—SCHEDULER
- 15.—PHYSICIST
- 16.—MATHEMATICIAN

Give number of object you wish to examine, or NIL to stop: 6

## (22) Documentation is available on the following templates:

- 1.—(>THING IS IN YOUR RANGE)
- 2.—(TRANSMIT TO YOUR FILTER-CENTER THAT +MESSAGE)
- 3.—(>PENETRATOR IS OUT OF YOUR RANGE)
- 4.—(TRY TO CHANGE GUIDER OF >FIGHTER TO >PENETRATOR)
- 5.—(>PENETRATOR IS DESTROYED)
- 6.—(>PENETRATOR HAS CHANGED ROUTE)
- 7.—(IS >PENETRATOR STILL IN YOUR RANGE)
- 8.—(GUIDE >FIGHTER TO >PENETRATOR)
- 9.—(STOP GUIDING >FIGHTER)
- 10.—(FIND A NEW GCI TO GUIDE >FIGHTER TO >PENETRATOR)
- 11.—(>FIGHTER HAS SIGHTED >PENETRATOR)
- 12.—(>FIGHTER UNABLE TO CHASE >PENETRATOR)
- 13.—(ARE SATURATED)
- 14.—(ARE NOT ECMED OUT BY >PENETRATOR)
- 15.—(CHECK FOR NEW PENETRATORS)

Give list of messages you wish to examine, or T for all, or NIL to stop: 8

Fig. 2—Trace of interactive browsing in ROSS and SWIRL  
(user's menu selections in boldface)

(23)      1.—ENGLISH DESCRIPTION  
              2.—SWIRL CODE  
              3.—BOTH

Type option, or NIL to stop: 1

(ask radar documentation for  
(guide >fighter to >penetrator) is

| *Sender: fighter or another radar handing over guidance of a fighter.*  
*Guides a fighter to a pen. Four cases arise: If the radar is destroyed*  
*it can do nothing. The fighter just follows unguided policy.*  
*If the fighter seeks guidance towards a penetrator that is not currently*  
*tracked by the radar, it tries to find another radar to guide the*  
*fighter. If this fails, it tells the fighter to follow unguided policy.*  
*If the radar is blinded, it tries to find another radar to guide the*  
*fighter. If this fails, it tells the fighter to follow unguided policy.*  
*Otherwise the radar calculates an intercept point of the fighter*  
*with the penetrator and tells fighter to vector to this point. |)*

Fig. 2—(continued)

knows about, and the user picks the one that he wishes to examine. At (22), the message templates of the selected object are displayed, and again the user selects one. Note how both the object-oriented structure of the simulation and the English-like nature of the message templates facilitate finding the exact part of the simulation the user wishes to examine. The object-oriented structure imposes a highly modular decomposition of the model, while the English-like templates make apparent exactly which behaviors are being modeled. At (23), the user chooses simply to see if the selected behavior embeds the right model. Had he selected option 2, ROSS would have put him into an editor with the code for the behavior.

## USING ROSS IN THE EMACS EDITOR

ROSS is an interactive programming environment, since it is implemented in Lisp. We have discussed several examples of how this interactivity assists the user in understanding his simulations. However, standard interactive programming environments have limitations, most of which stem from the "typewriter" mode of interaction almost all current software enforces. Even though most users run Lisp and ROSS on a terminal with a screen, the screen is useless. Like a typewriter-based terminal, only the current line can be submitted for evaluation. Previous lines cannot be referenced.

We sought to make ROSS a fully screen-oriented interactive environment in several ways. First, we wanted to make each command on the screen referenceable. Second, we wanted to give the user the ability to create multiple windows on the screen, each associated with a different process, such as an editor, a compiler, or a ROSS program. Finally, we wanted to allow users to "program by pointing." The user should be able to move a cursor to any expression on the screen and tell ROSS what he wants done with the expression.

To achieve these goals, we modified ROSS to run as a process under Emacs (Gosling, 1981), a customizable, screen-oriented editor. In this environment, ROSS is inside an Emacs window, and all forms typed go into the window's buffer. When the user types a carriage return, Emacs takes the form just typed and submits it to the ROSS process for evaluation, printing the value returned just below the form submitted. Thus, in simple interactions, the environment looks just like a standard Lisp session. However, because the user is in a screen-oriented editor, several new capabilities are available. Emacs commands can be used to move the cursor to and mark any previous form in the ROSS buffer. Emacs commands can then be used to apply several operations to the marked item, including the following:

- *Evaluate the item.* If the marked item is a legal Lisp or ROSS form, it can be resubmitted to the ROSS interpreter, and its value will be returned. The expression can be edited with any standard editor commands before reevaluation.
- *Compile the item.* If the marked item is the name or definition of a Lisp function or ROSS behavior, a few keystrokes will tell Emacs to create a Lisp compiler process, associate it with a new window, submit the marked definition to the process, and return the result. In this way, compilation can be much more incremental and interactive than many Lisp dialects permit.
- *Edit the item.* If the marked item is the name or definition of a Lisp function or ROSS behavior, the user can request Emacs to visit the file that contains the function or behavior. The user does not need to know where the function or behavior is defined. Emacs will create a window for the appropriate file and put the user in the file at the location of the specified definition. When the user is in this window, all the above capabilities are still at his disposal, in addition to the basic editor commands for deleting, adding, moving, and searching text. In particular, he can point to parts of definitions and ask for them to be evaluated. In this fashion, definitions can be rapidly and incrementally modified.
- *Reformat the item.* If the marked item is a Lisp expression, a single command can be used to reformat the expression in a more readable way. This facility is especially useful for cleaning up function or behavior definitions in their files.

The ROSS-Emacs facility reduces the model-builder's memory load by remembering the location of definitions. At the same time, it makes the examination and modification of models a rapid process by automatically handling many low-level programming details.

## TEXTUAL OUTPUT FACILITIES

We have discussed several features of ROSS that enable even naive users to locate, understand, and change complex simulation models. We now discuss some ways in which ROSS simplifies the analysis of simulation results, which too can be complex.

First, the interactive browsing in ROSS is easily extended to recording simulation events. For example, the recorder and historian we constructed for SWIRL work almost exactly like the browser. The recorder and historian are invoked by selecting option 8 in the SWIRL menu (item (24) in Fig. 3). Like the browser, the recorder and historian know all the objects and behaviors that comprise the SWIRL simulation and can guide the user through the code by displaying menus. The menus at (25) and (26) are similar to those at (21) and (22) in Fig. 2. However, when the user selects a behavior, the recorder and historian do not display the behavior, as the browser does; rather, they keep a record of each time that behavior is executed

(24) Select option:

- 1—Break into ROSS
- 2—Load compiled SWIRL
- 3—Load interpreted SWIRL
- 4—Recompile interpreted SWIRL files
- 5—Load a simulation environment
- 6—Run simulation with graphics
- 7—Run simulation without graphics
- 8—Activate historian and reporter
- 9—Browse or edit behaviors
- 10—Exit SWIRL & ROSS **8**

Specify file to record history: **RECORD**

Output to terminal (T or NIL): **T**

(25)

- 1.—PENETRATOR
- 2.—FIGHTER
- 3.—GCI
- 4.—SAM
- 5.—AWACS
- 6.—RADAR
- 7.—FILTER-CENTER
- 8.—COMMAND-CENTER
- 9.—FIGHTER-BASE
- 10.—TARGET
- 11.—MISSILE
- 12.—MOVING-OBJECT
- 13.—FIXED-OBJECT
- 14.—SCHEDULER
- 15.—PHYSICIST
- 16.—MATHEMATICIAN

Give list of objects in parentheses, e.g., (1 4 8), or NIL: **(1 2 4 5 6 7 9)**

(26)

PENETRATOR has the following message templates:

- 1.—(FLY TO >PLACE)
- 2.—(DROP >M MEGATON BOMBS EXPLODING AT ALTITUDE >H)
- 3.—(>RADAR IS NOW TRACKING YOU)
- 4.—(>RADAR IS NO LONGER TRACKING YOU)
- 5.—(EVADE)
- 6.—(RESCHEDULE YOUR NEXT SECTOR)
- 7.—(MAKE A RANDOM TURN)
- 8.—(MAKE A TURN >N DEGREES >DIRECTION)

Give list of message numbers to record (T for all): **T**

Fig. 3—Trace of interactive recording in ROSS and SWIRL  
(user's menu selections in boldface)

*[User and recorder interact to specify trace for other objects]*

Select option:

- 1—Break into ROSS
- 2—Load compiled SWIRL
- 3—Load interpreted SWIRL
- 4—Recompile interpreted SWIRL files
- 5—Load a simulation environment
- 6—Run simulation with graphics
- 7—Run simulation without graphics
- 8—Activate historian and reporter
- 9—Browse or edit behaviors
- 10—Exit SWIRL & ROSS 7

(27) Type number of ticks to run: 100

(28) 0.0 (AWACS3 LOOK FOR PEN3)  
0.0 (AWACS3 LOOK FOR PEN2)  
0.0 (AWACS3 LOOK FOR PEN1)  
0.0 (AWACS2 LOOK FOR PEN3)  
0.0 (AWACS2 LOOK FOR PEN2)  
0.0 (AWACS2 LOOK FOR PEN1)  
0.0 (PEN3 FLY TO (880.0 210.0))  
0.0 (PEN2 FLY TO (352.0 700.0))  
0.0 (GCI3 IS PEN2 STILL IN YOUR RANGE)  
0.0 (GCI2 IS PEN2 STILL IN YOUR RANGE)  
0.0 (PEN1 FLY TO (220.0 1085.0))  
492.891342 (AWACS1 PEN2 IS IN YOUR RANGE)  
492.891342 (AWACS1 ARE NOT ECMED OUT BY PEN2)

*[Simulation continues for 100 ticks]*

Select option:

- 1—Break into ROSS
- 2—Load compiled SWIRL
- 3—Load interpreted SWIRL
- 4—Recompile interpreted SWIRL files
- 5—Load a simulation environment
- 6—Run simulation with graphics
- 7—Run simulation without graphics
- 8—Activate historian and reporter
- 9—Browse or edit behaviors
- 10—Exit SWIRL & ROSS 10

Fig. 3—(continued)

during a simulation. An example of a partial record is shown at (28) in Fig. 3. It indicates both the simulation time at which the behavior was invoked and the message transmission that caused the invocation.

The recorder and historian allow simulation output to be tailored in several ways. First, because the specification of the objects and behaviors to record is done interactively, recording can be interwoven with running of the simulation (see (27) in Fig. 3). Thus, the user can rapidly change the parts of the simulation model he is examining. Second, the user has a great deal of control over which message transmissions are traced. In standard procedural languages, the only option available is to trace a function or not to trace it. Object-oriented languages like ROSS allow the user to trace all the message transmissions for all instances of a given class of object, or a specific message transmission for all instances of a class, or even particular types of message transmissions for specific instances of a class of objects. For example, in SWIRL one can focus on just the activities of *fighter1*, rather than all fighters.

### GRAPHICAL OUTPUT FACILITIES

A primary component of both the SWIRL and TWIRL simulations is their color graphics output facility. Although the graphics package is not an integral part of the ROSS language, we have developed a clean and direct interface between it and ROSS.<sup>3</sup> For each ROSS object, we define a symbol and color. Similarly, we define symbols for events that we wish to see graphically represented (e.g., communications between objects, military combat, radar detections). Each time the clock ticks, the graphics screen is updated to show the new locations of the objects and any new events that have occurred since the last clock tick. In a sense, the graphics package creates an animated movie of the simulation as it proceeds over time.

Graphics has provided us with a powerful tool for building and understanding simulations. Being able to watch the simulation as it is running enables the user to readily test out and verify behaviors, notice global interactions among the objects and events, zoom in on particular areas of interest, and determine the effects of alternative models and behaviors.

<sup>3</sup>SWIRL and TWIRL graphics were written in C by William Giarla. They were built upon graphics work previously developed at Rand for other projects.

### III. CONCLUSIONS

The ROSS language represents our first attempt to develop a relatively complete modeling environment. Ideally, such an environment can help users understand, reason about, and even design a wide variety of complex entities. In this report, we have discussed how ROSS facilitates the construction and viewing of models, through object-oriented programming techniques, English-like language syntax, interactive debugging and editing tools, and textual and graphical output facilities. Much work remains. We are presently investigating the Time Warp mechanism (Jefferson and Sowizral, 1982), a methodology for speeding up simulations by distributing them. In addition, we wish to develop modeling capabilities that go beyond simulation. We view a simulation as one of several kinds of models. Although other approaches to modeling are less well developed (de Kleer and Brown, 1983), they deserve attention because simulation alone enables users to answer surprisingly few questions about the behavioral characteristics of their models (Davis et al., 1982). To improve modeling as a reasoning and design tool, we must not only make it easier for users to ask questions of their models, we must greatly extend the range of questions they can ask.

## BIBLIOGRAPHY

Dahl, O-J., and K. Nygaard, "Simula—An Algol-Based Simulation Language," *Communications ACM*, Vol. 9, 1966, pp. 671-678.

Davis, M., S. Rosenschein, and N. Shapiro, *Prospects and Problems for a General Modeling Methodology*, The Rand Corporation, N-1801-RC, June 1982.

de Kleer, J., and J. S. Brown, "Assumptions and Ambiguities in Mechanistic Models," in D. Gentner and A. Stevens (eds.), *Mental Models*, Lawrence Erlbaum, Hillsdale, New Jersey, 1983.

Fain, J., D. Gorlin, F. Hayes-Roth, S. Rosenschein, H. Sowizral, and D. Waterman, *The ROSIE Language Reference Manual*, The Rand Corporation, N-1647-ARPA, December 1981.

Goldberg, A., and D. Robson, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

Gosling, J., *Unix Emacs*, Carnegie-Mellon University, Pittsburgh, 1981.

Hayes-Roth, F., D. A. Waterman, and D. B. Lenat, *Building Expert Systems*, Addison-Wesley, Reading, Massachusetts, 1983.

Hewitt, C., "Viewing Control Structures as Patterns of Message Passing," *Artificial Intelligence*, Vol. 8, 1977, pp. 323-364.

Jefferson, D., and H. Sowizral, *Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control*, The Rand Corporation, N-1906-AF, December 1982.

Kahn, K. M., *Director Guide*, Memo 482B, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1979.

Kiviat, P. J., R. Villanueva, and H. M. Markowitz, *The Simscript II Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1968.

Klahr, P., J. Ellis, W. Giarla, S. Narain, E. Cesar, and S. Turner, *TWIRL: Tactical Warfare in the ROSS Language*, The Rand Corporation, R-3158-AF, October 1984.

Klahr, P., and W. S. Faught, "Knowledge-Based Simulation," *Proceedings First Annual National Conference on Artificial Intelligence*, Palo Alto, California, 1980, pp. 181-183.

Klahr, P., D. McArthur, and S. Narain, "SWIRL: An Object-Oriented Air Battle Simulator," *Proceedings of the Second Annual National Conference on Artificial Intelligence*, Pittsburgh, 1982a, pp. 331-334.

Klahr, P., D. McArthur, S. Narain, and E. Best, *SWIRL: Simulating Warfare in the ROSS Language*, The Rand Corporation, N-1885-AF, September 1982b.

McArthur, D., and P. Klahr, *The ROSS Language Manual*, The Rand Corporation, N-1854-AF, September 1982.

Narain, S., D. McArthur, and P. Klahr, "Large-Scale System Development in Several Lisp Environments," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, West Germany, 1983, pp. 859-861.

Weinreb, D., and D. Moon, "Objects, Message Passing, and Flavors," *Lisp Machine Manual*, July 1981, 279-313.

**END**

**FILMED**

**3-85**

**DTIC**